Lab 4: Introduction to NASM, along with add and call

Taaruni Ananya | Session 008 | CRN: 15623

02/14/2025

Dr. Lan Gao

Introduction

Lab 4: Introduction to NASM, along with add and call uses the SNOWBALL server to delve into assembly code written for NASM and how it differs from previous labs' code. NASM is the "Netwide ASseMbler" for x86 CPUs, this uses this program and teaches how commonly it is used for programming in x86-based computers in assembly. This lab will cover the basics of NASM and compare Venus, gcc output, NASM, and Comment.

Purpose

The purpose of this lab is to continue practicing the use of SNOWBALL and script, along with terminal commands. Lab 4 is mainly focused on introducing code for NASM and its basics. From learning what NASM is, coding for it, to looking deeper with questions and answers. Lab 4 will also delve into teaching “mov”, “add”, and “call” commands, how code is stored in a text section, return value, and will compare output from various versions of assembly language code.

Main Objectives

- Learn the background, code, and how NASM works
- Compare various assembly code for various assembly language programs.
- Learn how code can be stored in a “text” section
- Learn “mov”, “add”, and “call” commands

Brief Explanation

NASM is the “Netwide ASseMbler” for x86 CPUs and is commonly used to program x86-based computers in assembly. NASM has similar commands as other programming but the code differs slightly; for example, the pound sign (“#”) is used for comments but NASM uses the semicolon (“;”).

Methodology/Procedure

The procedure starts with writing a program for NASM using the shortest version given in the instructions. This will then be copied on SNOWBALL, used for "nasm" command to assemble it, used with gcc to command link it, and finally be used to run the program. This will be repeated three times and then compared between each other to learn the differences and analyze what each difference means.

Documentation

AddTwoSum_64.asm

```
[tananya1@gsuad.gsu.edu@snowball ~]$ nasm -f elf64 AddTwoSum_64.asm
[tananya1@gsuad.gsu.edu@snowball ~]$ gcc AddTwoSum_64.o -o AddTwoSum_64
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64
```

AddTwoSum_64_pt2.asm

```
[tananya1@gsuad.gsu.edu@snowball ~]$ nasm -f elf64 AddTwoSum_64_pt2.asm
[tananya1@gsuad.gsu.edu@snowball ~]$ gcc AddTwoSum_64_pt2.o -o AddTwoSum_64_pt2
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt2
11
```

```
[tananya1@gsuad.gsu.edu@snowball ~]$ nasm -f elf64 -l AddTwoSum_64_pt2.lst AddTwoSum_64_pt2.asm
[tananya1@gsuad.gsu.edu@snowball ~]$ cat AddTwoSum_64_pt2.lst
     1                                  ; Assemble:   nasm -f elf64 AddTwoSum_64_pt2.asm
     2                                  ; Link:       gcc AddTwoSum_64_pt2.o -o AddTwoSum_64_pt2
     3
     4                                  ; Based on AddTwoSum_64.asm (by Kip Irvine)
     5                                  ; This is adapted for NASM.
     6
     7                                      extern  printf      ; We will use this external function
     8
     9                                      section .data       ; Data section, initialized variables
    10
    11 00000000 25640A00                mystr: db "%d", 10, 0   ; String format to use (decimal),
    followed by NL
    12
    13 00000004 0000000000000000        sum: dq 0
```

```
13 00000004 0000000000000000        sum: dq 0
14
15                                      section .text
16                                      global main
17                                  main:
18 00000000 B805000000                 mov  rax,5
19 00000005 4883C006                   add  rax,6
20 00000009 48890425[04000000]         mov  [sum], rax
21
22                                                     ; Now print the result out
23 00000011 48BF-                      mov   rdi, mystr   ; Format of the string to print
24 00000013 [0000000000000000]
25 0000001B 488B3425[04000000]         mov   rsi, [sum]   ; Value to print
26 00000023 B800000000                 mov   rax, 0
27 00000028 E8(00000000)               call  printf
28
29 0000002D B800000000                 mov  rax, 0
30 00000032 C3                         ret
```

```
0000530: b805 0000 0048 83c0 0648 8904 2538 1060  .....H..H..%8.`
```

```
18 00000000 B805000000                 mov  rax,5
```

AddTwoSum_64_pt3.asm

```
[tananya1@gsuad.gsu.edu@snowball ~]$ nasm -f elf64 AddTwoSum_64_pt3.asm
[tananya1@gsuad.gsu.edu@snowball ~]$ gcc AddTwoSum_64_pt3.o -o AddTwoSum_64_pt3
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt3
11
```

```
[tananya1@gsuad.gsu.edu@snowball ~]$ diff AddTwoSum_64_pt2.asm AddTwoSum_64_pt3.asm
1,2c1,2
< ; Assemble:     nasm -f elf64 AddTwoSum_64_pt2.asm
< ; Link:         gcc AddTwoSum_64_pt2.o -o AddTwoSum_64_pt2
---
> ; Assemble:     nasm -f elf64 AddTwoSum_64_pt3.asm
> ; Link:         gcc AddTwoSum_64_pt3.o -o AddTwoSum_64_pt3
28c28
<    mov  rax, 0
---
>    mov  rax, 3
```

```
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt2
11
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
0
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt3
11
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
3
```

Questions:

Part 1:

1. **Describe what this program does from the "main:" label to the end.**

   The first two lines of "main: " (`mov rax, 5` & `add rax, 6`) move 5 to the rax register and add 6 to the same register–with 5 + 6 resulting in rax holding the value 11. The next line (`mov [sum], rax`) stores rax's value into sum in memory. The fourth line (`mov [sum], rax`) resets rax to 0. The fifth line (`ret`) ends the program.

2. **What do you observe when you run it?**

   There is no output at all. After reexamining the code, it's obvious that there are values being stored in memory but nothing printed out to the output.

3. **Does the program work?**

   Yes, it seems to work since the program is started, commands are given, and the program is also ended with "`ret`".

Part 2:

1. **What do you observe? Does the program work? What does this program do that is different from the first one? (Describe what the assembly language commands do).**

   The second program prints out an output unlike the first one. While many of the commands are the same from the first one, there are four new commands. The first new line (`mov rdi, mystr`) brings string into rdi. The second new line (`mov rsi,` `[sum]`) moves the values in sum to rsi. The third new line (`mov rax, 0`) resets rax to 0. The fourth new line (`call printf`) calls printf and prints “11”. Yes, the program works because it is still calculated, printed, and ended.

2. **What do you observe in the file?**

   The new .lst file shows the assembly code, memory addresses, and what seems to be hex code.

3. **What do you observe there, and how does it relate to the .lst file? (Hint: look for the values B8 in the AddTwoSum_64_pt2.lst and b8 in the xxd output.)**

   The .lst and xxd pull up the program’s assembly and machine code. The values match up, proving what the assembly code is accomplishing but in the machine code side .

   (Screenshots included below with cmd + F):

```
0000530: b805 0000 0048 83c0 0648 8904 2538 1060  .....H...H..%8.`
```

```
18 00000000 B805000000                    mov   rax,5
```

4. **Do you observe any differences between this and AddTwoSum_64_pt2.asm? Use the "diff" command to show the differences between them, then explain what they are.**

   Yes, **AddTwoSum_64_pt2.asm** ends returning 0 while **AddTwoSum_64_pt2.asm** ends returning 3.

```
1,2c1,2
< ; Assemble:     nasm -f elf64 AddTwoSum_64_pt2.asm
< ; Link:         gcc AddTwoSum_64_pt2.o -o AddTwoSum_64_pt2
---
> ; Assemble:     nasm -f elf64 AddTwoSum_64_pt3.asm
> ; Link:         gcc AddTwoSum_64_pt3.o -o AddTwoSum_64_pt3
28c28
<    mov  rax, 0
---
>    mov  rax, 3
```

5. **What do you observe about the output from these two commands? Look up what a "return value" value is under Unix/Linux, describe what it is, and say how it relates to this lab.**

   After using “echo $” for each, **AddTwoSum_64_pt2.asm** returned 0 while **AddTwoSum_64_pt2.asm** returned 3. According to Google, “In Unix/Linux, a return value, also known as an exit status, is a numerical code returned by a process to its parent process (usually the shell) upon completion. It serves as an indicator of the process's success or failure. By convention, a return value of 0 signifies successful execution, while any non-zero value indicates that an error or issue occurred. “ This relates to his lab because we observe how and what the differences are in programs, machine code, and assembly code.

Key Code Observations

```
0000530: b805 0000 0048 83c0 0648 8904 2538 1060  .....H...H..%8.`
```

```
18 00000000 B805000000                    mov   rax,5
```

```
1,2c1,2
< ; Assemble:     nasm -f elf64 AddTwoSum_64_pt2.asm
< ; Link:                 gcc AddTwoSum_64_pt2.o -o AddTwoSum_64_pt2
---
> ; Assemble:     nasm -f elf64 AddTwoSum_64_pt3.asm
> ; Link:                 gcc AddTwoSum_64_pt3.o -o AddTwoSum_64_pt3
28c28
<    mov  rax, 0
---
>    mov  rax, 3
```

```
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt2
11
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
0
[tananya1@gsuad.gsu.edu@snowball ~]$ ./AddTwoSum_64_pt3
11
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
3
[tananya1@gsuad.gsu.edu@snowball ~]$
```